Dev and test set should come from same distribution.

Because dev set is what we use to evaluate model

Overfitting → high variance

underfitting → high bias

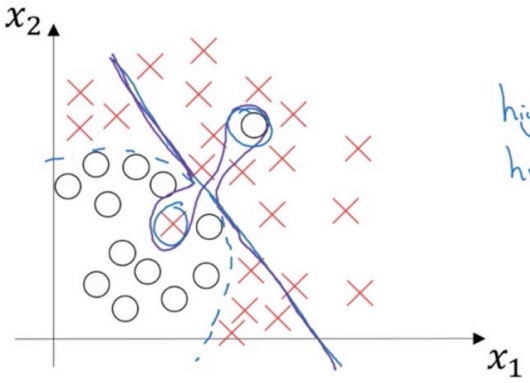
bias ↙

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

variance ↗

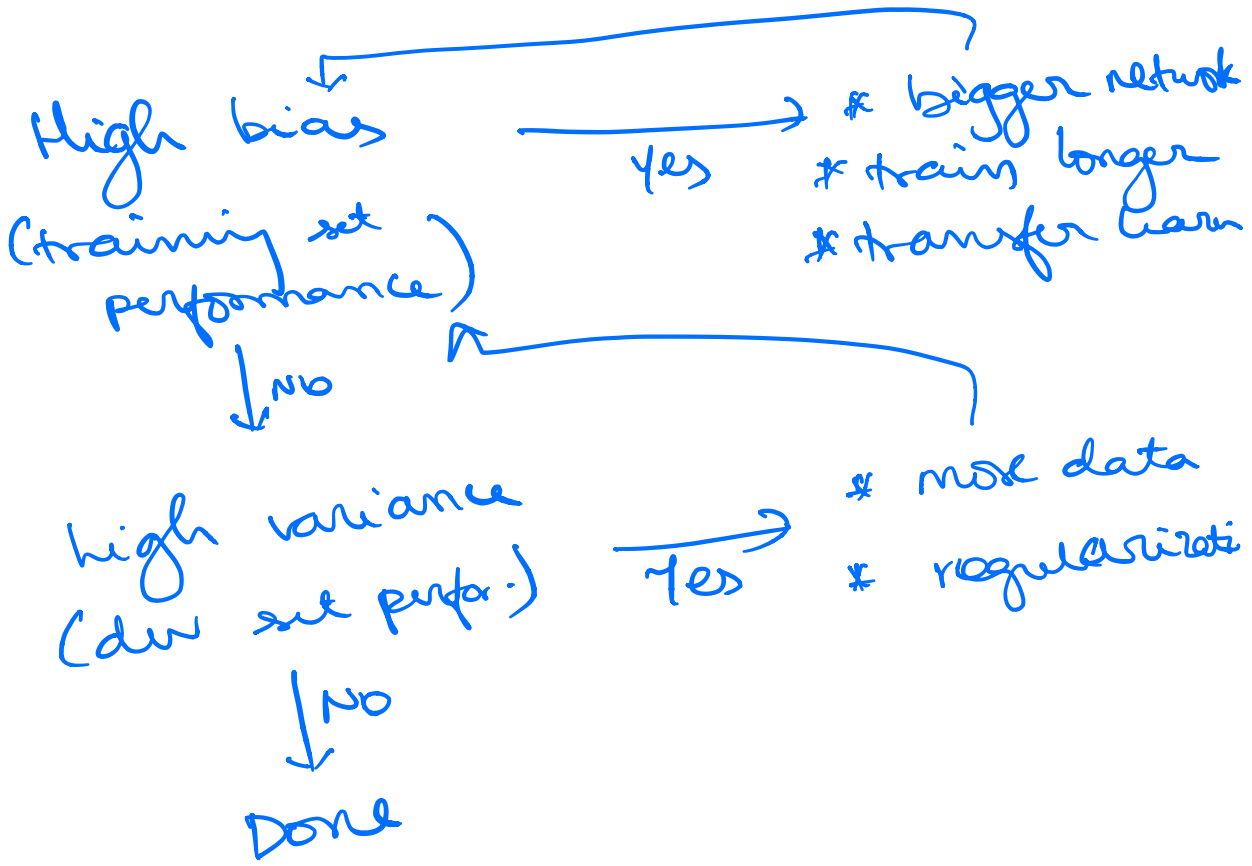
Human: ≈0%

Optimal (Bayes) error: ≈ 0%



high bias
high variance

Basic recipe:



We don't need to worry about bias - variance trade off as in DL, we can reduce one without affecting the other unlike algos in pre-DL era.

Regularization:

* prevent overfitting
* reduce variance

$$J(w,b) = \frac{1}{m}\sum \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 + \frac{\lambda}{2m}b^2$$

regularization parameter

$+\frac{\lambda}{2m}b^2$ — omit

L2 regularization:

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

L1 regularization: $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w| = \frac{\lambda}{2m}\|w\|_1$

L1 → w will be sparse (lots of zero) → less memory

$$\|w\|_n = \left(\sum w_i^n\right)^{1/n}$$

$$J(w^{[1]}, b^{[1]}, \dots) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|^2$$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

↳ frobenius norm

for backprop:

$$dw^{[l]} = (\text{usual}) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$

$$= w^{[l]} - \alpha\left(\text{usual} + \frac{\lambda}{m} w^{[l]}\right)$$

$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha(\text{usual})$$

reduces $w^{[l]}$

∴ L_2 also called weight decay

why regularization helps?

Since we have $\|\cdot\|_F$ added (for representing norm)

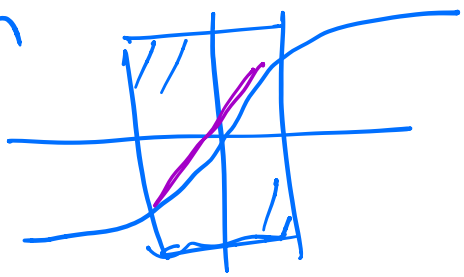
to $J()$ and we try to reduce value of J, this will try to reduce the values of w.

$\therefore$ if λ is large $\Rightarrow$ $w^{[l]} \approx 0$

Thus reducing w values and hence we get simpler network

another intuition:

Take tanh fn



If $\lambda \uparrow \Rightarrow w^{[l]} \downarrow$

$\therefore z^{[l]} = w^{[l]} A^{[l-1]} + b^{[l]}$

$\Rightarrow z^{[l]}$ is small

$\therefore$ values of $z^{[l]}$ will occupy small region of tanh. which is almost linear. This will prevent overfitting

Diff between L1 and L2:

1) L1 makes model sparse
→ model compression
→ feature selection
as coefficients become 0.

2) consider a case: $X: x_1, x_2$ and x_1 and x_2 are same

so: $y = x_1$ or $y = x_2$ is answer.

more generally : $y = ax_1 + bx_2$

where $a + b = 1$

L1 gives $y = x_1$ or $y = x_2$

L2 gives $y = 0.5x_1 + 0.5x_2$

So, when in real world, due to some system error we get only x_1 or x_2, model trained with L1 will suffer while model trained with L2 work.

∴ L2 models are more robust than L1.

Dropout regularization:

with some probability, remove some neuron

Implemented using 'inverted dropout'

Implementing dropout ("Inverted dropout")

Illustrate with layer $l=3$. keep-prob = 0.8 (0.2)

→ `d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob`

`a3 = np.multiply(a3, d3)` # a3 *= d3.

→ `a3 /= keep_prob` ← maintains expected value of a3 value of as same

50 units. ⇝ 10 units shut off

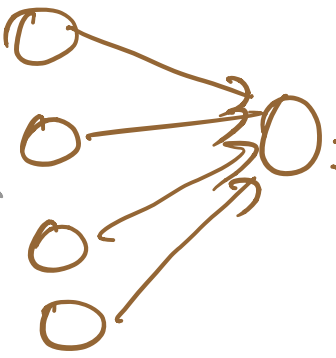
$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$

↑ reduced by 20%. Test

/= 0.8

no dropout when testing

Downside: J is not well defined



Since nodes are shut off randomly the node can't place all bets on one feature. So weights become shrunk

reduce overfitting:

* data augmentation
* early stopping

Normalizing:

subtract mean $\mu = \frac{1}{m}\sum x^{(i)}$

normalize variance

$\sigma^2 = \frac{1}{m}\sum x^{i} **2$

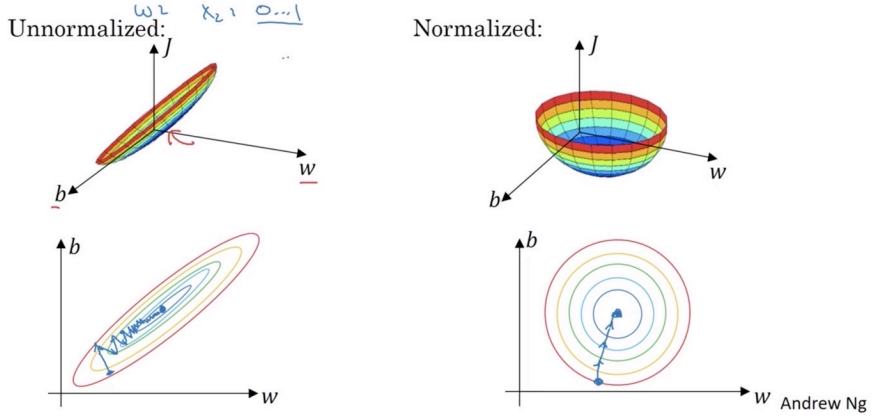
$x = \frac{x - \mu}{\sigma}$

Why normalize inputs?

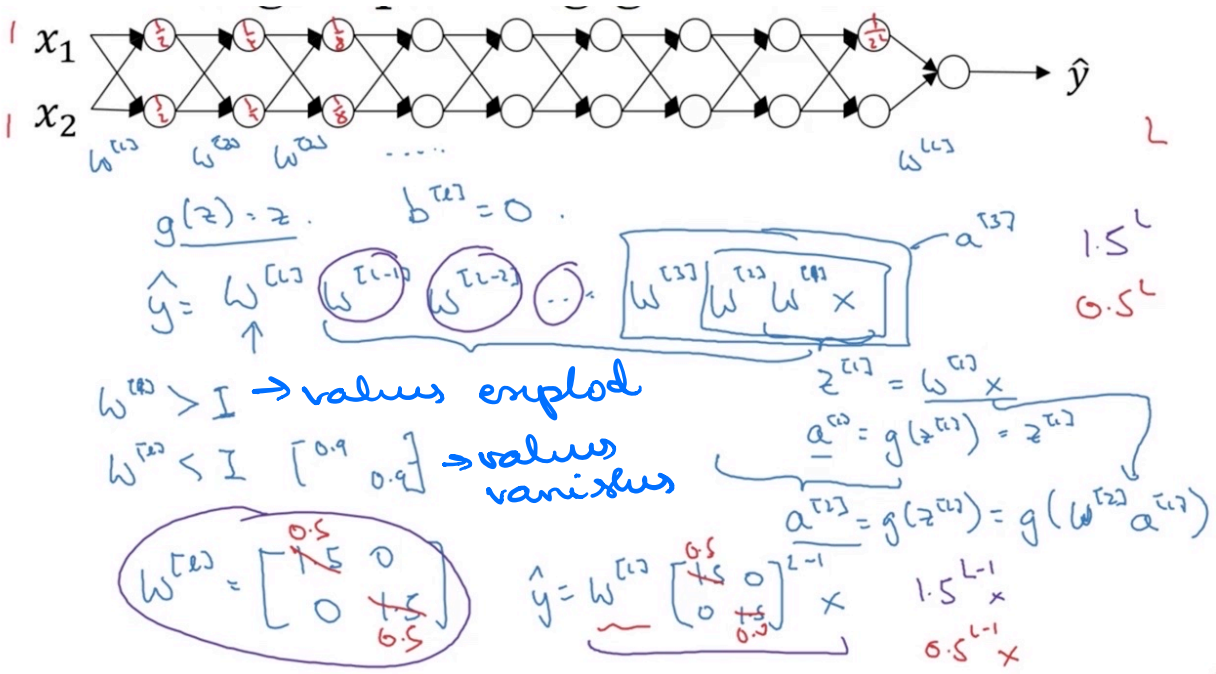
$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

w_1 x_1: 1....1000

w_2 x_2: 0...1



Vanishing / exploding gradients



Initialization:

variance = $\frac{x}{n}$

$w^{[l]} = np.random.rand( ) * np.sqrt\left(\frac{x}{n^{[l-1]}}\right)$

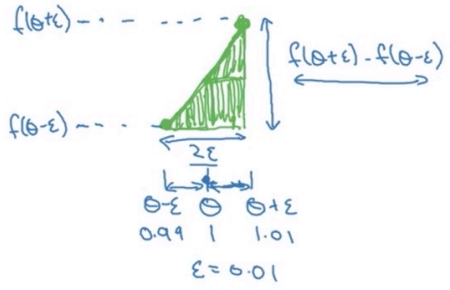
relu : $x = 2$

usually : $x = 1$ (tanh)

xavier : $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Checking your derivative computation

$f(\theta) = \theta^3$



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$$

$g(\theta) = 3\theta^2 = 3$

Approx error: 0.0001

(prev slide: 3.0301. error: 0.03)

$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ — $O(\varepsilon^2)$ (error): 0.01, 0.0001

$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$ — error: $O(\varepsilon)$ (error): 0.01

Andrew Ng

Gradient checking (Grad check)

$J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$

for each i:

$$\rightarrow d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i - \varepsilon, \ldots)}{2\varepsilon}$$

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i} \quad | \quad d\theta_{approx} \overset{?}{\approx} d\theta$$

Check $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$ $\approx$ 10^{-7} — great!

$\varepsilon = 10^{-7}$ $\qquad$ 10^{-5} $\rightarrow 10^{-3}$ — worry.

Mini batch gradient descent:

split training data into small batches

$X = \{ x^{\{1\}}, x^{\{2\}}, \ldots \}$

↑ one mini batch

mini batch t: $x^{\{t\}}$ - $y^{\{t\}}$

$x^{\{t\}}$: $(n_x, \text{size of mini batch})$

$y^{\{t\}}$: $(1, \text{size of mini batch})$

for $t = 1 \ldots$ no. of mini batches:

{ implement one step of gradient descent as if our training set is $x^{\{t\}}, y^{\{t\}}$

forward prop on $x^{\{t\}}$

compute cost $J^{\{t\}}$

back prop to compute gradient wrt $J^{\{t\}}, x^{\{t\}}, y^{\{t\}}$

update w, b

If mini batch size = m:

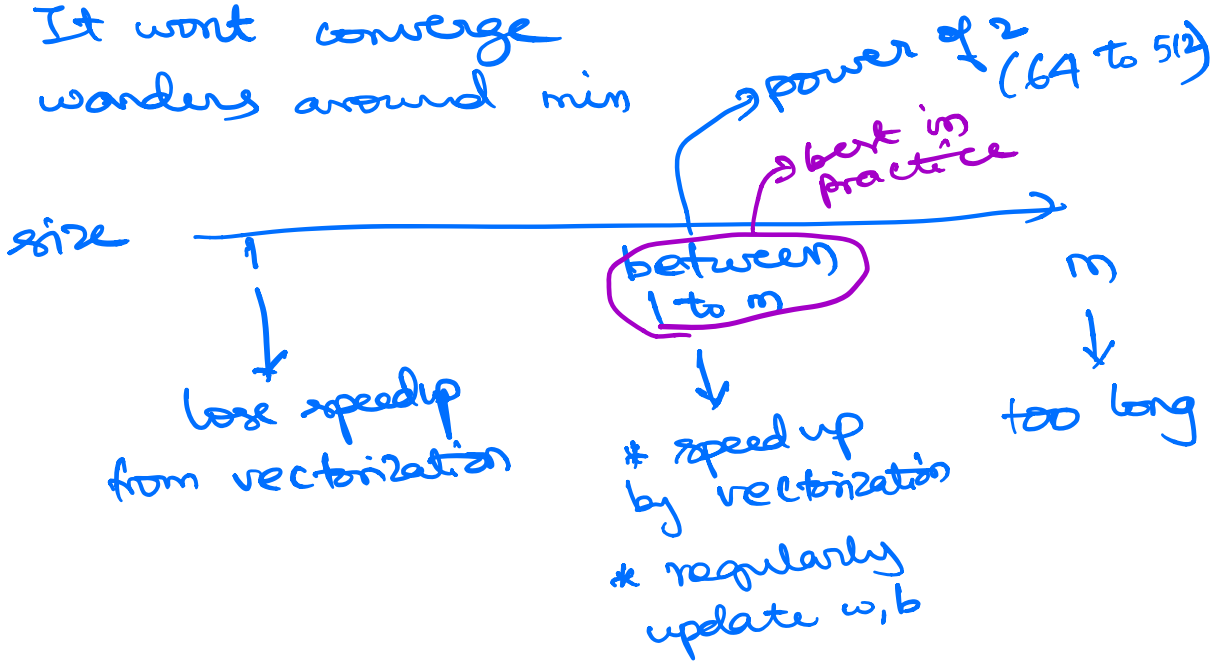
batch gradient descent

else mbs = 1:

stochastic gradient descent (every example is a mini batch)

It wont converge

wanders around min



Exponentially weighted moving average:

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

This is similar to approximately averaging over $\frac{1}{1-\beta}$ day's temp.

$\therefore \beta = 0.9 \Rightarrow \frac{1}{1-\beta} = \frac{1}{0.1} = 10$ days

$\beta = 0.98 \Rightarrow \frac{1}{1-\beta} = \frac{1}{0.02} = 50$ days

$\beta = 0.5 \Rightarrow \frac{1}{1-\beta} = \frac{1}{0.5} = 2$ days

$V_\theta = 0$

repeat:

get θ_t

$V_\theta = \beta V_\theta + (1-\beta)\theta_t$

} just store one array in memory

Bias correction:

Since V_θ is set to 0 initially, the starting part of the curve will be somewhat underfitting

∴ we can use $v_\theta = \frac{v_\theta}{1-\beta^t}$

So, in initial states, the denom will scale the values correctly as t becomes large, denominator approaches 1 and thus has no effect on v_θ values.

Gradient descent with momentum:

on iteration t:

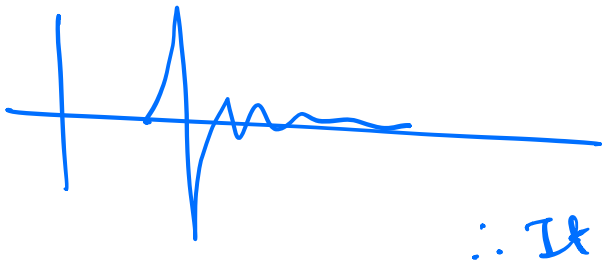
compute dw, db

$v_{dw} = \beta v_{dw} + (1-\beta) dw$

$v_{db} = \beta v_{db} + (1-\beta) db$

$$w = w - \alpha V dw$$

$$b = b - \alpha V db$$



averaging vertical axis will be almost zero

∴ It takes less oscillations along vertical axis while maintaing faster convergence

usually $\beta = 0.9$

$$Vdw = \beta Vdw + (1-\beta)dw$$

friction — velocity — acceleration

no need for bias correction as when $\beta = 0.9$, it will work correctly after ≈ 10 iteration.

Root mean square prop (RMS prop):

On iteration t:

$\varepsilon \Rightarrow$ small value so that denominator is not zero

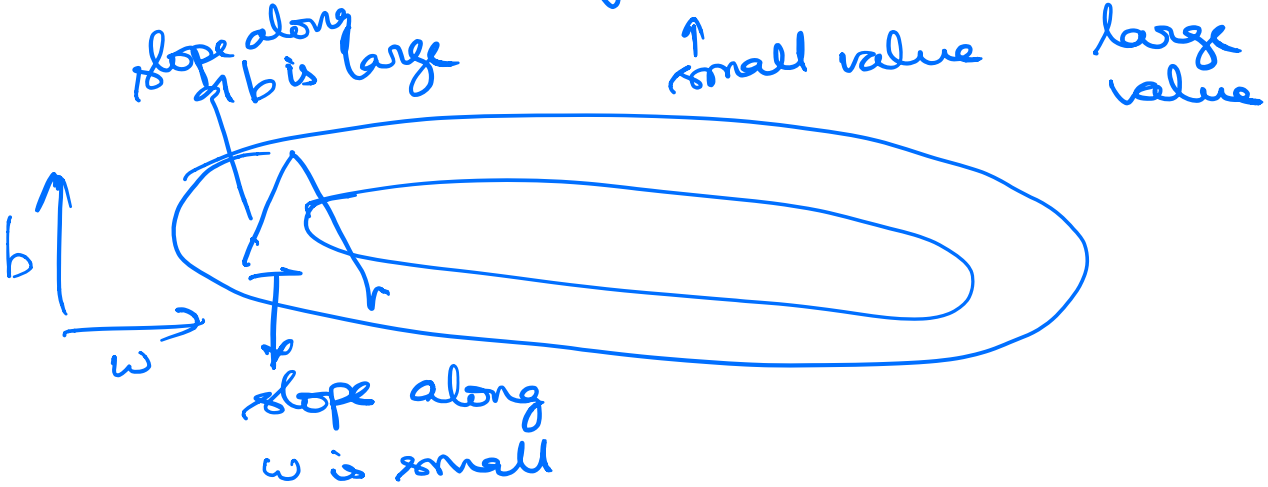
compute dw, db

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

$$w = w - \frac{\alpha\, dw}{\sqrt{S_{dw}} + \varepsilon} \qquad b = b - \frac{\alpha\, db}{\sqrt{S_{db}} + \varepsilon}$$

($\sqrt{S_{dw}}$: small value; $\sqrt{S_{db}}$: large value)



Adam optimization: (Adaptive moment estimation)

$V_{dw} = S_{dw} = V_{db} = S_{db} = 0$

on iteration t:

compute dw, db

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$

$V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$

$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$

$V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t}$

$V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$

$S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta_2^t}$

$S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$

Hyperparameters:

$\beta_1 = 0.9$

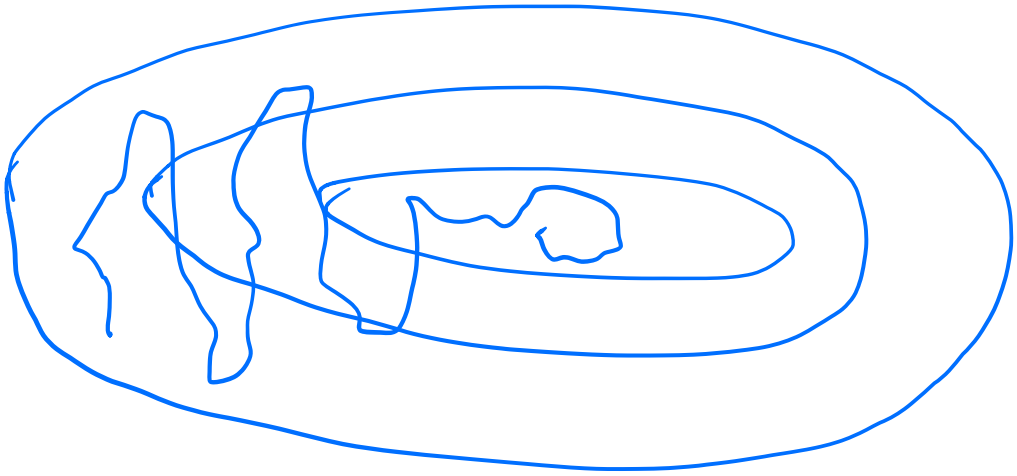
$\beta_2 = 0.999$

$\varepsilon = 10^{-8}$

$$w = w - \frac{\alpha \, Vdw^{corrected}}{\sqrt{Sdw^{corrected}} + \epsilon}$$

$$b = b - \alpha \frac{Vdb^{corrected}}{\sqrt{Sdb^{corrected}} + \epsilon}$$

Learning rate decay:



If we slowly reduce α per iteration:

initial steps: take larger steps

near convergance: take smaller steps

$$\alpha = \frac{1}{1 + \underset{rate}{decay} * \underset{num}{epoch}} * \alpha_0$$

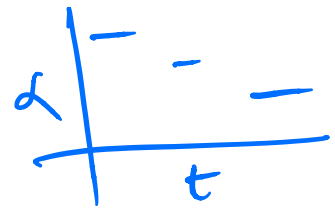
other ways:

formula

$\alpha = 0.95^{\text{epoch num}} \cdot \alpha_0$ (exponential decay)

$\alpha = \frac{k}{\sqrt{\text{epoch num}}}\ \alpha_0$ or $\frac{k}{\sqrt{t}}\ \alpha_0$

discrete decay

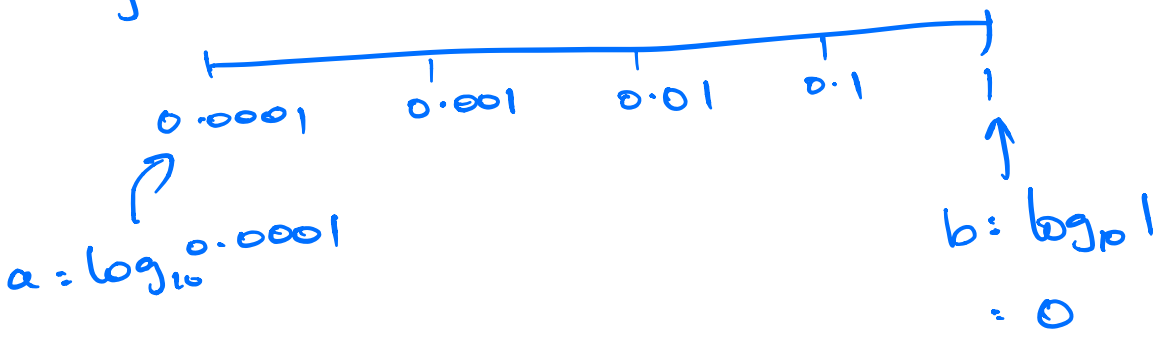


Tuning:

1. α
2. β, mini batch size, # hidden units
3. # layers, learning rate decay

1. consider random sampling of hyper parameters
2. consider coarse to fine search.

log scale for α:



$a = \log_{10} 0.0001$

$= -4$

$b = \log_{10} 1$

$= 0$

sample $r \in [a, b]$

and $\alpha = 10^r$

for β:

scale using $1-\beta$

$\beta = 0.9 \ldots\ldots 0.999$

$1-\beta = 0.1 \ldots\ldots 0.001$

$a = -1 \qquad b = -3$

$r \in [-3, -1]$

$1-\beta = 10^r$

$\beta = 1 - 10^r$

Batch normalizing:

→ makes model robust

→ has slight regularization effect

$$\mu = \frac{1}{m} \sum_i z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i \left(z_i^{[l](i)} - \mu \right)$$

$$z_{norm}^{[l]} = \frac{z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

This will make mean 0 and var 1

if we want something different

use it for further calculations $\tilde{z}^{[l]} = \gamma z_{norm}^{[l]} + \beta$

γ, β: learnable parameters of the model

size: $(n^{[l]}, 1)$

used GD to update γ and β as we do for w & b

if: $\gamma = \sqrt{\sigma^2 + \epsilon}$ $\beta = \mu$

$\Rightarrow \tilde{z} = z$

covariate shift:

$X \longrightarrow Y$

if x's distribution changes, retrain algo

while testing:

we won't have μ, σ as we'll work on one sample.

So use exponentially weighted average of μ for all mini batches and use it when testing

Softmax Layer: (multiclass classification (Softmax regression))

activation fn:

$$t = e^{(z^{[L]})}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i} \left(\text{or} \frac{t_i}{\sum t_i} \right)$$

y : (C, 1) ↑ # classes

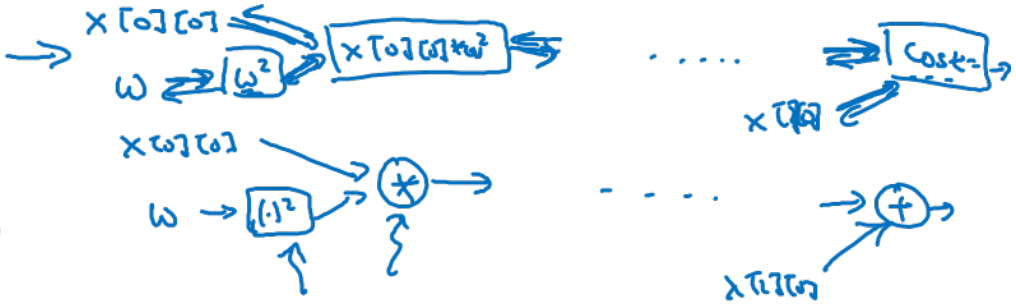
loss fn : $-\sum_{j=1}^{C} y_j \log \hat{y}_j$

$J = \frac{1}{m} \sum \text{Loss}$

Back prop:

Initialize $d2^{[L]} = \hat{y} - y$ and continue back prop.

Code example



```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]    # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

Andrew Ng